Control Flow Aspects of an Algebraic Approach to Compiler Generation (Mathematical Studies of Information Processing)

AUTHOR(S):

CHRISTIANSEN, HENNING; JONES, NEIL D.

数理解析研究所講究録
454巻 1982年 110-137

110

First draft:

Control Flow Aspects of an Algebraic Approach to Compiler Generation

Henning Christiansen
Neil D. Jones
Computer Science Department
Aarhus University
8000 Aarhus C, Denmark

This paper is an exposition of a new approach to compiler generation which was described at the symposium on "Mathematical Studies of Information Processing" held in May 1981 at the Kyoto Research Institute for Mathematical Sciences. It describes research currently in progress; we intend eventually to publish a more polished and complete version including correctness proofs.

Abstract

An algebra-based method for automatically generating compilers is described. Its input is the semantics of a programming language expressed in "imperative semantics", a restricted form of denotational semantics. Its output is a compiler which maps program parse trees into a target code consisting of flow charts with "computed goto", from which machine code may be generated by macro expansion or traditional code generation techniques. Imperative semantics allow natural semantic definitions for a wide variety of programming language features, and may be regarded as a powerful compiler-writing language. The method generates target programs and compilers which are both efficient and easily ported.

1 INTRODUCTION

An idealized compiler generation scheme could have the form given in Figure 1. Much progress in this direction has already been made. The well-understood theory of context-free languages has been put to practical use in several systems based on automatic parser generation including XPL [McK70], BOBS [EKM79] and YACC [Joh74]. These syntax-directed compiler-writing systems however leave all semantic questions such as symbol table management and code generation to the user, only providing a mechanism to call "semantic routines" when grammatical phrases are recognized. The formalism of attribute grammars [Knu68] provides methods to handle some semantic aspects, notably those with a "static" character such as symbol tables and the composition of target code sequences; however control flow aspects are not handled at all. Compiler-writing systems based on these include MUG2 [Gan77], DELTA [Lor75], HLP [Räi77], NEATS [RiM81].

Automatic generation of code generators has been difficult due to questions of how to describe the form and effect of machine code instructions. However some progress has been made in this direction including [Cat79] and [Don79]. Many of the remaining problems appear to be of an engineering nature.

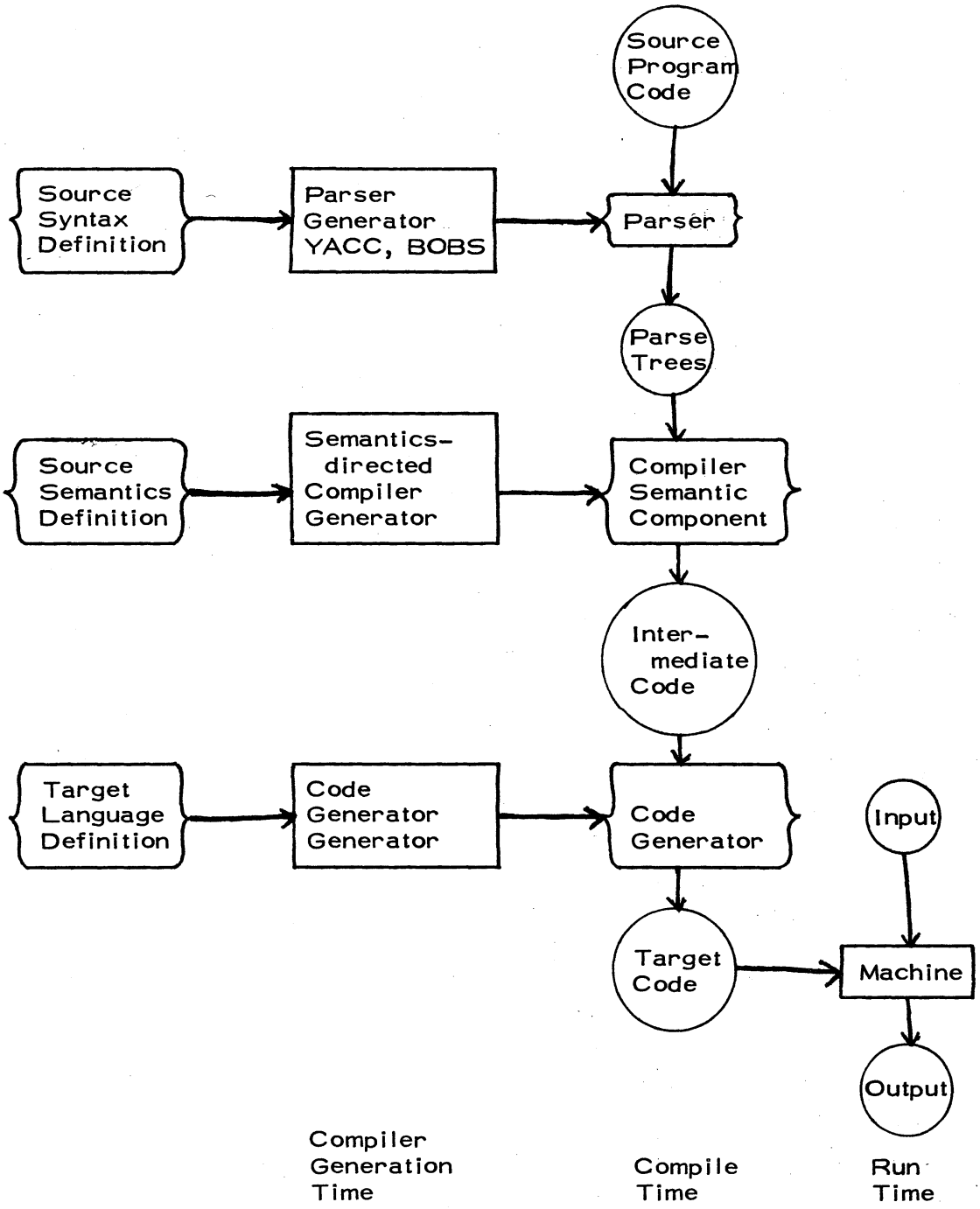


Figure 1. Idealized Compiler Generation Scheme.

Earlier Work in Semantics-Directed Compiler Generation

Formal definitions of the semantics of programming languages are becoming more and more widely accepted (e.g. [Mos74], [Gor79] and [ADA80]) for a variety of reasons. Ideally it should be possible to start with a precise definition of the semantics of a language and automatically produce from this a correct compiler. We have termed this goal "semantics-directed compiler generation" in [Jon80].

An important step in this direction is the two-volume Milne and Strachey [MiS76] development (by hand) of a compiler from a specific language definition given in denotational semantics. A series of alternative semantics are defined; each new semantics is closer to machine code than its predecessor, and each is proven equivalent to the original semantics. The transformations are complex and the equivalence proofs even more so (actually much more complex than writing a compiler).

This development is exceedingly complex, and appears extremely difficult to systematize and generalize as would be necessary to do for compiler generation. Nonetheless it has been the basis for many other works including [Bjø77], [Gan80], and [Ras80]. Wand describes in [Wan80a] and [Wan80b] methods using combinators to transform the right sides of semantic equations into forms closer to machine code.

Another approach is to view the right sides of the semantic equations as an intermediate language, to be further processed or translated. This approach is implicitly taken in [JoS80] and is one way to use SIS [Mos79].

An interesting class of methods is based on partial evaluation or mixed computation. Suppose $\mathcal{S}$ is a semantic definition, π is a source program, and i is a run-time input. Then $(\mathcal{S}(\pi))i$ represents first finding the meaning $\mathcal{S}(\pi)$ of program π (typically an input-output function), and then applying that to input i. $\mathcal{S}$ is normally represented by a syntactic object, so "compilation" may proceed by attempting to evaluate (or reduce) $\mathcal{S}(\pi)$ as much as possible in the absence of input i. This reduced form is then the target program.

Partial evaluation has been studied by Sandewall [San75] among others. Ershov and Turchin use it in contexts where $\mathcal{S}$ is defined operationally (by ALGOL68 and REFAL programs, respectively, see [Ers78] and [Tur80]). The Semantics Implementation System (SIS) of Mosses uses similar ideas, where $\mathcal{S}$ is a denotational semantics expressed in a lambda-calculus extension called DSL[Mos79].

Efficiency is a problem with this class of methods since compilation involves complicated transformations within the language which is used to write the semantic definition. The target programs are of course expressed in the same language.

Another approach is due to Ganzinger [Gan80], who develops a series of operations which allow a limited class of denotational semantics to be transformed

into attribute grammar form. The operations are, however, rather complex and seem to be of limited generality. He also describes an approach for separation of the dynamic and static (= invariant with respect to runtime input) components of the resulting attribute grammars, and ideas concerning code generation.

Yet another class of methods is based on algebraic semantics. These seem to have considerable promise, and are described in later sections.

Outline of this paper

In section II we give a brief review of algebraic semantics, its application to compiler correctness proofs, and an overview of the new approach, emphasizing its relation to earlier work. Section III introduces IS, a family of algebras we use for writing imperative semantics, and gives an example consisting of a set of semantic equations without a specific model. We discuss general requirements which we impose on all models which guarantee that program execution may be viewed as performing a sequence of elementary actions which manipulate an (as-yet-unspecified) "store". Next, an "interpretation" of IS is defined; this includes a concrete definition of the store and specifies the effects of elementary actions on the store and control point. Section III ends with the interpretation for the example introduced earlier, and a discussion of the generality of the class of imperative semantics.

Section IV describes the target programs - essentially flow charts with "computed goto", whose boxes contain elementary actions. Section V introduces a "compiling algebra" C, and explains how this is used to generate compilers. A correctness theorem is stated but not proved, asserting that the generated compilers always produce target programs whose semantics is faithful to their source programs. Thus individual compiler correctness proofs (e.g. [MiS76], [ADJ79]) are not needed.

Section VI describes C in detail, the compiler generation function cg: IS → C, and the standard interpretation of C. Finally, section VII contains practical remarks, mentions a pilot implementation in PROLOG, and ends with conclusions and directions for future work.

II AN ALGEBRAIC VIEW OF DENOTATIONAL SEMANTICS

Three major methods have been used to provide formal definitions of programming language semantics: axiomatic, operational and denotational. To date axiomatic semantics have proved to be difficult to apply to a broad spectrum of languages, since features such as recursion with local and global variables, aliasing and pointer variables complicate the logical formalisms which are used; further, it seems very difficult to develop program execution methods from axiomatic specifications.

Definitions via operational semantics ([Luc69], [Oll75]) have suffered from the opposite problem, of being biased towards a specific implementation strategy. Further they tend to be rather complex, typically involving an extremely complex total execution state. (Some of these faults may however be alleviated by the new "structured operational semantics" now being developed by Plotkin [Plo81]).

Denotational semantics begins with the so-called "denotational assumption". In [ScS71] this is expressed as:

> "The semantical definition is syntax directed in that it .... transforms each language construct into the intended operations on the meanings of its parts."

This leaves two questions open: "What is a meaning?" and "How does one operate on meanings?". Dana Scott has provided a rich theory of domains which are sufficient to express both the meanings of syntactic objects and effective computations involving these meanings [Sco76]. While we use these foundations, we will not dig into them. A syntactic object's meaning is typically called a denotation.

A natural and general framework for operating on meanings is provided by many-sorted algebras. We describe briefly only what is needed for this work; more details and background may be found in [ADJ77] or [Mor73].

Definition A signature consists of

1. a collection S of sorts (examples: integer, boolean, program, expression)
2. a collection of operator symbols w, each with its arity
   $w: s_1 \times s_2 \times \ldots \times s_n \to s_0$ where each $s_i \in S$ is a sort and $n \geq 0$.

A constant is an operator symbol of arity $w: \to s_0$, that is $n = 0$.

A many-sorted algebra consists of

1. a signature $(S, \{w_i\})$
2. a set A_s for each $s \in S$ (called the carrier of sort s)
3. an operation $\overline{w}: A_{s_1} \times \ldots \times A_{s_n} \to A_{s_0}$ for each operator symbol w of arity $s_1 \times \ldots \times s_n \to s_0$.

Example: A Term Algebra for Syntax

Consider the (infinite) context-free grammar with productions

program	::=	exp	
exp	::=	$x_1 \mid x_2 \mid \ldots$	(variables)
	|	0 | 1 | 2 | ...	(constants)
	|	exp + exp	
	|	$\lambda x_1.\text{exp} \mid \lambda x_2.\text{exp} \mid \ldots$	(abstractions)
	|	$\text{exp}_1\ (\text{exp}_2)$	(application)

This may be seen as a 2-sorted algebra:

Signature

Sorts: S = {program, exp}

Operator Symbols: one for each production

Arities: map righthandside nonterminals to lefthandside sort, e.g. the arity of exp ::= λx_2.exp is exp → exp.

Carriers

A_{exp} = {π | π is a well-formed parse tree with root label exp}

$A_{program}$ = similar

Operations

These build parse trees from their subtrees according to the productions. For example, let $\pi_1, \pi_2 \in A_{exp}$ and w be exp::= exp + exp. Then

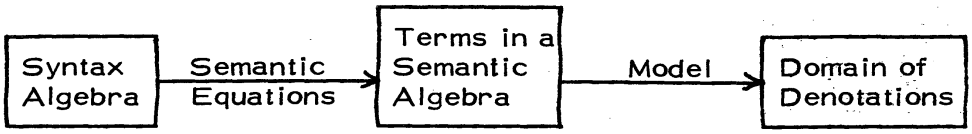
$\bar{w}(\pi_1, \pi_2)$ =  is in A_{exp}

□ Example

The algebra above is known as a term algebra, since each of its elements may be viewed as a term (= tree) built up from constants by use of the other operator symbols. Here, each production A → terminal* is a constant.

Denotational Semantics

A denotational semantics is naturally factorable into two parts, one mapping parse trees into a semantic algebra and one mapping the semantic algebra further into a domain of denotations. References: [Gor79], [Ten76], [Sto77], [ADJ77].



The semantic algebra which has been most used in denotational semantics (explicitly or implicitly) is the natural term algebra describing expressions of the λ-calculus. Stoy for example [Sto77] (following [Sco76]) defines the LAMBDA notation, and models its operator symbols (abstraction, application, conditional, etc.) by operations on the domain P_ω. The notation can become complex to say the least - for example semantic equations for the first two productions of the syntax given above might be

$$\mathcal{E}[\![\text{variable}]\!] = \lambda\rho\lambda k.\ k(\rho\,[\![\text{variable}]\!])$$
$$\mathcal{E}[\![\text{exp}_1+\text{exp}_2]\!] = \lambda\rho\lambda k.\ \mathcal{E}[\![\text{exp}_1]\!]\rho(\lambda a.\mathcal{E}[\![\text{exp}_2]\!]\rho(\lambda b.k(a+b)))$$

Alternative Semantic Algebras

In a series of papers ([Mos78], [Mos80], [Mos81]), Mosses has been advocating the use of semantic algebras which better reflect the intuitive concepts naturally present in computation-sequencing, production and consumption of values, binding, etc. The benefits include not only more readable semantic definitions but also greater modularity, and seem well-suited to generalizable proofs of compiler correctness. As we shall see they are also well-suited to compiler generation.

Most of Mosses' work is based on abstract data types, so the semantic algebras of [Mos78] and [Mos80] are the initial algebras obtained by factoring a term algebra by an equivalence relation induced by a set of equations, as described for example in [ADJ78]. However [Mos81] uses what amounts to a term semantic algebra (his "syntax") modeled by Scott-style function domains.

We find the latter approach more natural for compiler generation, since a generated compiler is after all a syntactic object. In any case one wants both in practice: equations are needed so that one may manipulate (e.g. compute with) the terms appearing in semantic definitions without having to think of the complexities involved in an underlying domain theory; and a model is needed to show that the equations are not inconsistent. The only question is where to begin.

Compiler Correctness via Algebraic Semantics

A series of papers have been written concerning expression and proof of compiler correctness in the framework of algebraic semantics ([Gau80], [BuL69], [Mor73], [ADJ79], [Mos80]). We now describe two of them and relate them briefly to the new method; deeper comments will be made after the new method has been more fully described.

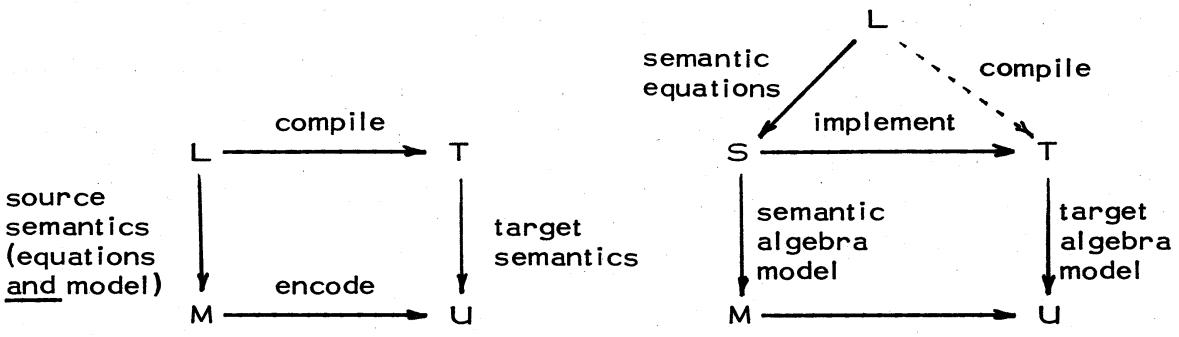


Figure 2. Compiler Correctness Diagrams

In both parts of Figure 2 L is a source language regarded as a G-algebra, where G is the context-free grammar defining the abstract syntax of L. M is a set containing denotations (meanings) of pieces of source programs. T is a collection of target programs, and U contains their denotations.

In the left part of figure 2 (from [Mor73] and [ADJ79]) one is given a compiling function L → T, a source semantics L → M and a target semantics T → U. The problem is to show that the meanings of compiled programs faithfully represent the meanings of the original source programs. This is done by introducing an "encoding" function mapping source meanings to their representations via target meanings and proving commutativity of the diagram (and perhaps injectivity of "encode" to show that T and U were not badly chosen).

By the "denotational assumption" the source semantic function L → M may be expressed as a homomorphism to an algebra derived from M. This is obtained by associating with each syntax operator of L a corresponding derived operator on M. Both [Mor73] and [ADJ79] assume "compile" is given homomorphically via derived operators on T. Thus a compound program is compiled by combining the results of compiling its parts, using combination operators present in T. The target semantics T → U is also specified homomorphically, so in the diagram each corner becomes a G-algebra, with M, T, U all derived.

A syntax algebra is "initial" in its category, meaning that there exists a unique homomorphism from it to any other algebra in the category. Consequently if it can be shown that "encode" is a homomorphism, then commutativity and so compiler correctness follows at once from initiality of L. [Mor73] and [ADJ79] use this method.

From this viewpoint it is natural to view the carriers of T as containing "pieces" of target programs, and to map each operator symbol of L into an operation for "pasting together" these pieces of target programs in an appropriate way. This view is quite clear in the (rather different) target algebras of [Mor73], [ADJ79] and [JoS80]. A note: correctness in [JoS80] is not shown by commutativity, but by a more operational argument involving equivalence of phrases in L.

The right part of the diagram, from [Mos80], introduces an explicit semantic algebra S as an intermediary between L and M, as described in the previous section. S is defined independently of L. Further, [Mos80] requires that each of S and T be an abstract data type - that is the initial algebra $T_{\Sigma,\mathcal{E}}$ for the category of all algebras with signature Σ which satisfy a set $\mathcal{E}$ of equations between Σ-terms (S and T always exist but may be trivial). The "implement" function should be a correct implementation of one abstract data type by another as defined, for example, in [ADJ78].

The "compile" function is simply the result of composing the original semantic equations with "implement". Note: models M and U are needed to show the nontriviality of S and T, but are not involved directly in the proof that "implement" correctly implements S by T.

This approach has significant advantages over the first. If S is well chosen it can serve to express the semantics of a wide variety of source languages; proposals for S may be found in [Mos80] and [Mos81]. Correctness of "implement" need be shown only once; thereafter a correct compiler for any language L whose semantics is expressible via S can automatically be constructed by composition with "implement".

Typically this can be done at a symbolic level, so a set of semantic equations for L can be transformed into a new set of equations, a "compiling semantics" which maps parse trees directly into elements of the target algebra.

Compiler Generation via Algebraic Semantics

Our method most resembles that of [Mos80] but has some important differences. To begin with our approach is model-based (like [Mos81] but not [Mos80]). Two term algebras are used: IS (for "imperative semantics"), a simple semantic algebra playing a similar role to S in figure 2, and a new "compiling algebra" C. In effect T in figure 2 has been split into two parts as in figure 3.

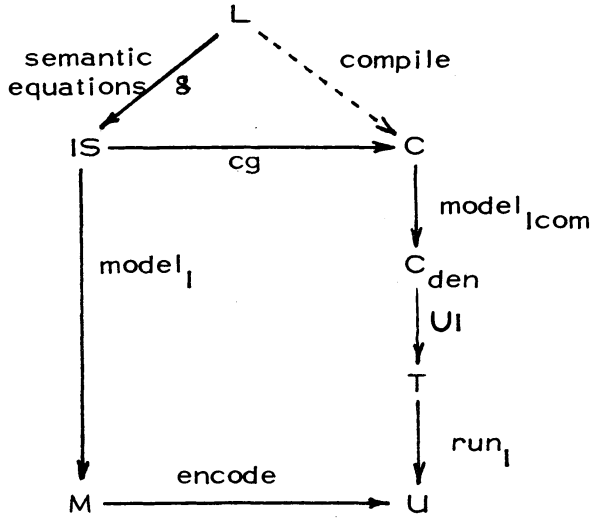


Figure 3. Compiler Generation Scheme.

The separation between C and T corresponds to the traditional distinction between "compile-time" and "run-time". It allows for great flexibility in the choice of C since it is not required that each element of C have a concrete meaning in the target semantic model U (as must be true of T in figure 2). In this particular application "cg" maps elements of IS into "compiling functions" rather than pieces of target program. If such a function is applied to a partially constructed target program, it will cause the generation of additional target code and/or the updating of a "compile-time state". Other choices of C and its compile-time model might allow expression of compile-time loops, multipass compilation, etc., which seems difficult to express in the frameworks of figure 2.

Another consequence of the approaches of figure 2 is that T must have a sufficiently rich structure so that derived operations modelling the syntax operators, of L can be defined. This in turn implies that corresponding operations must be defined on U. This can lead to some complex and unnatural semantic rules. For example (from [Mor73], p. 149),

> "To compute in a flowchart sewn together from pieces is to compute by turns in the pieces, jumping back and forth as often as one pleases at the stitches."

We see no good reason for introducing such complexities into the semantics of a target language which is intended to model, for example, machine code instructions. Separation of C and T allows program construction to occur at compile time, and target program execution to be expressed by a more natural and low-level semantics (run_I in figure 3) independent of L, IS and C.

The left side of figure 3 indicates that the semantic equations map parse trees into terms of a semantic algebra IS, which are then modeled via "$model_I$" in an algebra M of source denotations. In this paper IS will be very simple, specifying only control flow; consequently target code denotations will be almost the same as source denotations, so M=U (exception: labels in U represent continuations in M).

The semantics of a source program in L may be thought of as the execution of a sequence of elementary "actions", each operating on a "store" which might typically hold values of variables, control stacks, etc. The elementary actions will be represented syntactically by the constants (plus parameters) appearing in IS. Their semantics will be specified by the "interpretation" I seen in "$model_I$" - this defines the set of all stores, and the effects of the elementary actions on the store and on the control point (all this will be precisely defined in the next section).

We will define (in section VI) a "compiler generation" function cg which will map terms of the semantic algebra IS into terms of the compiling algebra C. Composition of this with the given semantic equations yields a "compiling semantics", a set of equations mapping parse trees into terms in C. The compound map $L \to C \to C_{den}$ will map a parse tree into its denotation, namely a compiling function

which upon application to a partially compiled target program will cause target code for the parse tree to be added to the target program.

Note that $L \to C \to C_{den}$ is also a semantic definition; further C is a subalgebra of IS and Icom is an interpretation specifying compile-time stores and store transformations. Thus a common framework is used for source semantics and the generated compilers. Denotations of programs in L will be target programs, so $T \subseteq C_{den}$.

The target code I can loosely be described as "flowcharts plus computed gotos". The exact instruction set produced depends on the constants appearing in the semantic equations for L, and their semantics is specified by the interpretation I. In fact, the target language semantics function run_I is also determined by I.

Correctness can be established by showing that the diagram between IS and U commutes for every interpretation I. This implies that the function "compile" will take any parse tree in L into a term of C which, if executed with the compile-time interpretation Icom, will produce a target program whose semantics (when interpreted according to I) is identical to that of the given parse tree. Note: we will see that in practice compiling can be done without explicitly building the C term, so in spite of appearances this is not really an intermediate-language method.

III IMPERATIVE SEMANTICS

We now define a rather simple semantic term algebra IS (for "imperative semantics"). An imperative semantics of a programming language will be given in two parts. One is a set of semantic equations mapping source program parse trees into terms of IS; these terms will be constructed from elementary actions and atomic values (integers, strings, etc.) by the operator symbols of IS. The remainder of the semantic definition is an interpretation I which defines the domains of "stores" and "final answers", and gives meanings to the elementary actions. More precisely, I specifies a certain homomorphism from IS into a continuation-based domain of denotations.

The models of IS we consider are restricted so that term evaluation may be thought to consist of performing a series of elementary actions upon a store (the exact sequence is of course determined by the initial store). Imperative semantics may thus be thought of as a formalization of (continuation-based) "store semantics" as used in [MiS76] and by many others. One extension is that we factor the definition so that store-related details appear only in the interpretation and not in the semantic equations. Further, the combination of IS and I is capable of expressing a great many disciplines of program control flow.

Given an imperative semantics, it is easy to translate parse trees into a target code T. This is essentially a flow chart language with "computed goto", where the boxes in the flow charts contain elementary actions. Oddly enough, the

flowcharts contain no explicit branching structure, being simply collections of labeled linear instruction streams. All nonsequential control flow (e.g. looping, conditional branching, calls and returns) is handled by passing labels as parameters to elementary actions, which can save and fetch them in and from the store as well as branch to them. The counterpart in IS of these labels are the "delayed action sequences" allowed as parameters to elementary actions.

In spite of the close connections between IS and sequential, machine-like execution, the class of programming languages whose semantics can be naturally defined is surprisingly large. This will be discussed further at the end of this section.

Table 1. Signature of IS

Indices: $\Delta = \{a\} \cup \{s \mid s \text{ is an atomic sort}\}$

Sorts:

ans	answers (values of entire source programs)
e	elementary actions, each with a source σe in Δ^* $(= \Delta \cup \Delta \times \Delta \cup \ldots)$
p	parameters, each with a target $\tau p \in \Delta$
a	actions
s	various atomic sorts, e.g. integers, boolean, string

Operations:

a	<=	skip	empty action
	|	$a_1; a_2$	sequencing - do a_1, then a_2
	|	e	elementary action with empty source
	|	$e(p_1, \ldots, p_n)$	elementary action with parameters. Requirement: $\sigma e = \tau p_1 \times \ldots \times \tau p_n$
	|	fix L: a	recursively-defined action (where L is a variable)
p	<=	con	constant of atomic sort as parameter. p has target τp = sort of con
	|	a	open action as parameter. Target τp = a
	|	a; L	closed action as parameter. Target τp = a, and L is a variable from a fix.
ans	<=	execute(a)	perform an action sequence

Notation

IS is presented in a style close to that of [Mos80] or [Mos81]. Operator symbols are written in "mixfix" notation (called "distributed-fix" in [Gog78]). This is a generalization of prefix, infix and postfix notation: operator symbols can be distributed freely around and between operands. Further, the arity of an

operator is indicated by the notation

$$S_0 \Leftarrow f(S_1, \ldots, S_n)$$

where f has arity $s_1 \times \ldots \times s_n \rightarrow s_0$.

One unusual construction is used (resembling one used by Mosses). Each elementary action with parameters will have certain "type" requirements on the sorts of its parameters (e.g. find(x) requires x to be a variable). These requirements are expressed by equipping each elementary action with a "source" which is a sequence of sorts, and each parameter with a "target" which is the sort of its value.

Explanation

An action can be the null action "skip", an elementary action (possibly with parameters), or can consist of two subactions to be done in sequence. Further, an action can be defined recursively via fix. This may be conceived of via infinite "unrolling", i.e. fix L:a is expanded replacing each free occurrence of L in a by fix L:a and iterating. Note that L may only be used to close an action parameter, i.e. it may only appear in "tail recursive" position. This will make it possible to translate IS terms into flowchart code.

Intuitively an action specifies for each store a series of transformations to be performed and the "final answer" execute(a) is obtained by applying a's transformations to the initial store. Each store transformation is specified by an elementary action, with or without parameters. The mapping from elementary action constants in the IS algebra to store and control transformations is given by the interpretation I.

An action appearing as a parameter can be thought of as a data value representing a "delayed action". This value may be put into the store by an elementary action (e.g. pushed onto a "return stack"), and can at some later time be fetched from the store and executed. An elementary action may change the flow of control dynamically by selectively activating delayed actions, thus giving the effects of loops, conditional branches, subroutine or coroutine calls and returns, etc.

An open action parameter, e.g. a in ...e(a, 13)... represents a series of elementary actions which after execution will normally be followed by the actions after e(a, 13) - the second "..." in this case. A closed action parameter a;L is similar; except that after a's actions are performed, execution will continue with the actions specified in the smallest enclosing fix L:a'.

Actions as parameters are represented in the standard model by continuations, and in the target programs by labels. Otherwise, actions are modeled by continuation transformers (all described later in this section).

Table 2. An Example Imperative Semantics $\mathcal{S}$

Atomic sorts: integer, variable

Semantic Equations:

$\mathcal{P}[\![\text{program}]\!] = \text{execute}(\mathcal{E}[\![\text{program}]\!])$

$\mathcal{E}[\![\text{con}]\!] = \text{load(con)}$

$\mathcal{E}[\![\text{var}]\!] = \text{find(var)}$

$\mathcal{E}[\![\text{exp}_1 + \text{exp}_2]\!] = \mathcal{E}[\![\text{exp}_1]\!]; \mathcal{E}[\![\text{exp}_2]\!]; \text{plus}$

$\mathcal{E}[\![\underline{\text{if}}\ \text{exp}_1\ \underline{\text{then}}\ \text{exp}_2\ \underline{\text{else}}\ \text{exp}_3]\!] = \mathcal{E}[\![\text{exp}_1]\!]; \text{choose}(\mathcal{E}[\![\text{exp}_2]\!], \mathcal{E}[\![\text{exp}_3]\!])$

$\mathcal{E}[\![\text{exp}_1\ (\text{exp}_2)]\!] = \mathcal{E}[\![\text{exp}_1]\!]; \mathcal{E}[\![\text{exp}_2]\!]; \text{apply}$

$\mathcal{E}[\![\lambda x.\, \text{exp}]\!] = \text{save(bind(x); } \mathcal{E}[\![\text{exp}]\!]; \text{ return)}$

Examples of use of $\mathcal{E}$:

$\mathcal{E}[\![x + (\underline{\text{if}}\ y\ \underline{\text{then}}\ 7\ \ \underline{\text{else}}\ z) + 8]\!]$

= find(x); find(y);
choose (load(7), find(z));
plus; load(8); plus

$\mathcal{E}[\![(\lambda x.\, xx)(\lambda y.\, y)7]\!]$

= save(bind(x); find(x); find(x); apply; return);
save(bind(y); find(y); return);
apply;
load(7); apply

Intended Interpretation of Example Semantics

Table 3 contains an interpretation I precisely defining the meanings of the elementary actions used in table 2. Here we describe them informally, partly to demonstrate the naturalness of an imperative semantics. Formal definitions of an interpretation I and its induced model follow table 3.

The store consists in this case of a value stack and an environment mapping variables to values. The intended effect of performing $\mathcal{E}[\![\text{ex}]\!]$ is to push the value of "exp" on the stack, leaving the environment unchanged. It is easy to define meanings for "load" and "find" doing this, and "plus" should thus pop 2 values and push their sum (as is expressed by the first three equations of table 3).

$\mathcal{E}[\![\underline{\text{if}} \ldots]\!]$ first pushes the value of exp_1, and "choose" pops this and activates $\mathcal{E}[\![\text{exp}_2]\!]$ or $\mathcal{E}[\![\text{exp}_3]\!]$. After the appropriate $\mathcal{E}[\![\text{exp}_i]\!]$ actions are performed, execution will continue with the first action following $\mathcal{E}[\![\underline{\text{if}} \ldots]\!]$ (this follows from the way parameters are modeled – see the definition below of the model induced by an interpretation).

$\mathcal{E}[\![\lambda\, x.\, exp]\!]$ will push a certain "delayed action" onto the stack. The equation for $\mathcal{E}[\![exp_1 (exp_2)]\!]$ resembles that for $\mathcal{E}[\![exp_1 + exp_2]\!]$ but "apply" is essentially different from "plus": it saves the action sequence following "apply" on the stack and then activates the delayed action which must be the value of $\mathcal{E}[\![exp_1]\!]$ (i.e. the value of some abstraction $\lambda\, x.\, exp$). This will bind x to the value of exp_2, evaluate the body exp in the updated environment (leaving the result on the stack), and finally return to the control point and environment which were saved by "apply".

Table 3. Interpretation of Example Semantics

Domains:

Store = Env × Value* — typical element: st = (e, s)

where Env = [Variable → Value] — typical element: e

Value = N + Cont × Env — typical elements: a, b

Cont = [Store → Answer] — typical element: c

Answer = Value

Elementary Actions:

(θload) con c (e, s)	= c (e, con·s)	[push constant]
(θfind) var c (e, s)	= c (e, e[[var]]·s)	[push value of var]
(θplus) c (e, b·a·s)	= c (e, (a+b)·s)	[pop a, b, push a+b]
(θchoose) (c_1, c_2) c (e, a·s)	= if a then c_1(e, s) else c_2(e, s)	
(θsave)(c_1)c (e, s)	= c (e, (c_1, e)·s)	[save action on stack]
(θapply) c (e, a·(c_1, e_1)·s)	= c_1 (e_1, a·(c, e)·s)	[activate saved action]
(θreturn) c (e, a·(c_1, e_1)·s)	= c_1 (e_1, a·s)	[return]
(θbind) var c (e, a·s)	= c(e{a/var}, s)	[update env. binding]

Final Continuation:

c_∞ (e, a·s) = a — [answer = stack top]

Initial Store:

St_0 = (empty environment, empty stack)

Note: The "value" stored for an abstraction consists of a continuation and an environment, essentially the "closure" typically used in λ-calculus implementations (the environment is needed so binding is static rather than dynamic). The functionalities of θload, etc. are found in the definition of "interpretation".

Omission

For the sake of simplicity in the rest of this paper we will no longer treat actions containing fix or variables, or their implementations. No essential problems arise, and a fuller treatment will be given in a later version of this paper. Briefly, fix and variables can be handled by introducing a semantic environment in model$_1$: $e \in [Var \to Cont]$, binding a new continuation to L wherever fix L:a is encountered, and accessing e wherever a variable reference is found in a. Similarly, a "compile-time environment" binding variables to runtime labels must be added to the compile-time store in the domain of denotations model$_{com}$ of the compiling algebra C.

Interpretations and induced homomorphisms

We now define an "interpretation" I and the homomorphism it induces more formally. The underlying ideas are simple enough:

1. We only consider homomorphic images of IS in which actions are modeled by continuation transformers, the semicolon operator is modeled by composition of continuation transformers, and a few other natural requirements described in the second definition below.

2. Thus any particular image of IS can be specified by defining the store and answer sets and the denotations of the constants (e.g. "plus", "if") which are used. This is an interpretation.

Definition Let $\mathcal{S}$ be a set of IS semantic equations. An interpretation of $\mathcal{S}$ is a 5-tuple $I = (Store, Answer, \theta, c_\infty, st_0)$ satisfying 1-6 below. First, let the set of continuations be $Cont = [Store \to Answer]$ ($[A \to B]$ is the set of continuous functions from domain A to domain B).

1. θ associates with each parameter p appearing in $\mathcal{S}$ a domain Dp such that
   a) if $\tau p = a$ then $Dp = Cont$ (action parameters are interpreted as continuations)
   b) if $\tau p = s$ for an atomic sort s, then Dp = a domain corresponding to s (e.g. sort "integer" can map to $\{0, 1, 2, \ldots\}$)

2. Let e be an elementary action with empty source appearing in $\mathcal{S}$. Then
   $$\theta e \in [Cont \to Cont]$$

3. Let $e(p_1, \ldots, p_n)$ appear in $\mathcal{S}$. Then
   $$\theta e \in [Dp_1 \times \ldots \times Dp_n \to [Cont \to Cont]]$$

4. If p is a constant parameter of atomic sort appearing in $\mathcal{S}$, then $\theta p \in Dp$ (e.g. the constant "13" of IS maps to the integer thirteen)

5. $c_\infty \in$ Cont (the final continuation)
6. $st_0 \in$ Store (the initial store)

□ definition

Remark

2-4 imply that θ assigns values to all the generators (i.e. constants) of IS which appear in $\mathcal{S}$. This implies I can be extended to a homomorphism on all of IS (restricted to $\mathcal{S}$) in the following way:

Definition Let $\mathcal{S}$, I be as above. The model induced by I is defined as follows:

Model$_I$

Carriers:

A_{ans} = Answer, $A_a = [\text{Cont} \to \text{Cont}]$
A_e = $[\text{Cont} \to \text{Cont}]$ if e has empty source
A_e = $[Dp_1 \times \ldots \times Dp_n \to [\text{Cont} \to \text{Cont}]]$ if $e(p_1, \ldots, p_n)$ appears in $\mathcal{S}$
A_s = Dp if $\mathcal{S}$ contains a parameter p with atomic sort s
A_p = Cont → Dp for each parameter p appearing in $\mathcal{S}$

Operations:

a	⇐ skip	$(\theta a)\, c$	=	c	[no action]
	| $a_1; a_2$	$(\theta a)\, c$	=	$(\theta a_1)(\theta a_2 c)$	[sequencing]
	| e	θa	=	θe	
	| $e(p_1, \ldots, p_n)$	$(\theta a)\, c$	=	$\theta e(\theta p_1 c, \ldots, \theta p_n c)\, c$	[each parameter is given "next" c as argument]
p	⇐ con	$(\theta p)\, c$	=	θ con	
	| a	$(\theta p)\, c$	=	θ ac	[action parameter is linked to "next" c]
ans	⇐ execute(a)	θ ans	=	$(\theta a) c_\infty\, st_0$	

How general are imperative semantics?

To date semantics definitions for five minilanguages embodying various programming language features have been constructed, and may be found in [Chr81]. (These use an earlier nonalgebraic version of imperative semantics, but are easily converted to the present formalism.) They include

1. A simple language similar to those of [ADJ79] and [Mos80].
2. One with recursive procedures.
3. One with coroutines.
4. One with nested blocks and gotos.
5. The lambda calculus.

The definitions are easy to read and were not difficult to construct. Further, it appears straightforward to combine the various features into a single imperative semantics. Some "nontraditional" approaches had to be used when writing the semantics due to the limited domains allowed. For example, procedure names are "traditionally" bound to continuation transformers [Cont → Cont]. In order to replace this by Cont, it was necessary to add a data structure to the store - the familiar "return address stack" - holding the continuation used to exit from a call statement. For another example, the usual fixpoint definition of label semantics had to be done in a different way, but the result is not especially difficult to understand or write.

In fact, the kind of reasoning one uses to write semantics in imperative form is quite similar to that used in ordinary compiler design - just at a higher level. At the very least, one could regard IS as a rather powerful compiler-writing language, from which the semantics components of compilers can automatically be produced.

On the other hand imperative semantics are certainly limited - for example they lack the produced and consumed values of [Mos80] (corresponding to traditional usage of expression continuations), and allow no semantic variables beyond those found in fix L:a. Another limitation is that no form of static/dynamic analysis and separation is done. This could lead to undesirable features such as runtime symbol tables (as are seen in our small example).

None of these limitations seems to be intrinsic to our method; they are simply other aspects of compiling which we expect can be comfortably fit into the paradigm of figure 3, and plan to work on in the near future. The main point is that an algebraic approach using operators natural for expression of primitive computing concepts, and a clear formalization of "compile-time" versus "run-time" seem to provide a very appropriate framework in which to develop more sophisticated compiler generation methods.

IV THE TARGET LANGUAGE

Table 4. Syntax of T

program	::=	[0: stream$_0$ 1: stream$_1$ ... k: stream$_k$]	
stream	::=	instr | stream; instr	
instr	::=	goto (destination)	
	|	e	elementary action
	|	e(p_1,...,p_n)	ditto with parameters
p	::=	atomic value	
	|	l	label of stream origin
l	::=	integer	origin labels
destination	::=	l + integer	stream origin + displacement

Note that the syntax of T contains neither conditional branching nor the "computed goto" referred to earlier; these will be handled semantically, using label parameters and the saving of labels in the store. Terms in IS are easily translated into T code by hand - for example:

Code⟦x + (if y then 7 else z) + 8⟧
= [0: find(x); find(y); if(1,2); plus; load(8); plus
1: load(7); goto(0+3)
2: find(z); goto(0+3)]
} both go to "plus ..." in stream 0

Code⟦(λx.xx)(λy.y)7⟧
= [0: save(1); save(2); apply; load(7); apply
1: bind(x); find(x); find(x); apply; return[;goto(0+1)]
2: bind(y); find(y); return[;goto(0+2)]]

Note: the code above is actually the output which our generated compiler will produce for the semantics of table 2. The goto's in brackets are redundant, but are produced since the compiler does not "know" that in this interpretation "return" always ignores its continuation.

Semantics of T

For brevity we only outline this, since no difficult or new concepts are involved. Actually two semantics are defined. One is for use in the statement of correctness, and yields M = U, i.e. identical source and target denotations. This is however unsuitable for implementation purposes since it involves storing higher-order objects (namely continuations) as data objects in the store. Thus an equivalent first-order semantics is defined in which all continuations are represented by the target code labels they correspond to. We give a few characteristics of the first semantics; extension to the second is straightforward.

1. The semantics run_1 of T is induced by an interpretation
$I = (Store, Answer, \theta, st_0, c_\infty)$.

2. Given a program $\pi = [0: stream_0 \ldots k: stream_i]$,
   a) each $stream_i$ and instruction is mapped to a continuation transformer
   b) each label $l \in \{0,\ldots,k\}$ is mapped to a continuation $\rho(l)$ with the aid of a runtime environment ρ: Labels → Cont
   c) ρ is defined recursively by a fixpoint
   d) the program is mapped to the "final answer" $\rho(0)\ (st_0)$.

3. An instruction which is an elementary action e without parameter is mapped to its value θe as specified by interpretation I.

4. Suppose $e(p_1,p_2)$ takes an integer parameter p_1 and an action parameter p_2 in IS. Then in the target code instruction, $e(p_1',p_2')$ will take an integer parameter $p_1' = p_1$ and a runtime label parameter p_2'. The semantics of $e(p_1',p_2')$ is exactly $\theta e(p_1',\rho p_2')$, i.e. the label parameter is interpreted as the continuation which is its valuation by ρ.

V COMPILER GENERATION

We now return to figure 3. The compiling algebra C is simply a subalgebra of IS, restricted to an interpretation Icom which specifies the compile-time store and the elementary actions needed to generate target code. The only actions we need (in the absence of fix and variables) are the following. More formal descriptions will be found in the next section.

addcode(ins)	-	add a new instruction "ins" to the end of the instruction stream currently being generated
neworig	-	establish the origin of a new stream (used when an action parameter is processed)
oldorig	-	re-establish the old origin previously in use; further, save the new origin in a compile-time stack p^*
push(atom)	-	push an atomic parameter on the compile-time stack p^*
$addcode_n(ins)$	-	add instruction $ins(p_1,\ldots,p_n)$ to the current stream, where $p_1,\ldots,p_n$ are parameters found on the compile-time stack
goback	-	end the current stream with an instruction "goto(destination)" to transfer control back to the stream previously in use

Table 7 contains a "compiler generation" homomorphism cg: IS → C, mapping each operator of IS to a derived operator of C which is built from the above by semicolon. For any action a of IS, cg(a) will be a term of C whose denotation in C_{den} will be a function which, if given a program, will add to it instructions to perform "a" at runtime.

Table 5 contains the result of composing the original semantics of table 2 with "cg", yielding a "compiling semantics". Table 6 contains the target program resulting from application of the compiling semantics to a source program.

Table 5. Compiling Semantics Generated from Table 2

$\rho^C[\![\text{program}]\!]$ = execute ($\varepsilon^C[\![\text{program}]\!]$)

$\varepsilon^C[\![\text{con}]\!]$ = push(con); addcode$_1$(load)

$\varepsilon^C[\![\text{var}]\!]$ = push(var); addcode$_1$(find)

$\varepsilon^C[\![\text{exp}_1+\text{exp}_2]\!]$ = $\varepsilon^C[\![\text{exp}_1]\!]$; $\varepsilon^C[\![\text{exp}_2]\!]$; addcode(plus)

$\varepsilon^C[\![\underline{\text{if}}\ \text{exp}_1\ \underline{\text{then}}\ \text{exp}_2\ \underline{\text{else}}\ \text{exp}_3]\!]$ =

$\varepsilon^C[\![\text{exp}_1]\!]$;
neworig; $\varepsilon^C[\![\text{exp}_2]\!]$; goback; oldorig;
neworig; $\varepsilon^C[\![\text{exp}_3]\!]$; goback; oldorig;
addcode$_2$(choose)

$\varepsilon^C[\![\text{exp}_1(\text{exp}_2)]\!]$ = $\varepsilon^C[\![\text{exp}_1]\!]$; $\varepsilon^C[\![\text{exp}_2]\!]$; addcode(apply)

$\varepsilon^C[\![\lambda x.\ \text{exp}]\!]$ = neworig;
push(x); addcode$_1$(bind);
$\varepsilon^C[\![\text{exp}]\!]$;
addcode(return);
oldorig;
addcode$_1$(save)

Table 6. An Application of Table 5

Source Program π = (if x then 7 else (y+1)) + z

Compiling Semantics Output (a sequence of elementary compile-time actions):

$\rho^C[\![\pi]\!]$ = execute(push(x); addcode$_1$(find);
neworig; push(7); addcode$_1$(load); goback; oldorig;
neworig; push(y); addcode$_1$(find);
push(1); addcode$_1$(load); addcode(plus);
goback; oldorig;
addcode$_2$(if);
push(z); addcode$_1$(find); addcode(plus))

Value of ρ^C under Compile-time Interpretation Icom

[0: if(1,2); find(z); plus
1: load(7); goto(0+1)
2: find(y); load(1); plus; goto(0+1)]

Table 7. Compiler Generation Function cg: IS → C

IS Operators	Derived Operators on C
a <= skip	skip
| a_1; a_2	cg(a_1); cg(a_2)
| e	addcode(e)
| $e(p_1,\ldots,p_n)$	cg(p_1); ... cg(p_n); $addcode_n$(e)
p <= con	push(con)
| a	neworig; cg(a); goback; oldorig;
ans <= execute(a)	execute(cg(a))

Correctness statement

Referring to Figure 3 we can now state correctness of the method. Proof of a closely related result may be found in [Chr81]. The compiling semantics of for example table 5 becomes $\mathcal{S}' = \mathcal{S} \circ cg$ in the current notation.

Theorem Let $\mathcal{S}$ be an imperative semantics (i.e. a homomorphism from a syntax algebra to terms of IS). Then for any interpretation I of $\mathcal{S}$ and any source program parse tree π,

$$model_I(\mathcal{S}\pi) = run_I(model_{Icom}(cg(\mathcal{S}\pi)))$$

This theorem asserts that for any semantic definition $\mathcal{S}$ written in terms of IS, the "compiling semantics" $\mathcal{S}' = \mathcal{S} \circ cg$ is a correct compiler. This completely avoids the need for individualized (and difficult) compiler correctness proofs such as those of [ADJ79], [BuL69], [Gau80], [McP67], [MiS76], [Mor73], [Mos80] and [Wan80b].

Generality of the method follows from the expressive power of IS combined with the fact that interpretation I may be chosen freely. However it is clearly desirable to expand IS to allow for more powerful and less operationally oriented semantic definitions.

VI A COMPILING ALGEBRA C

The elementary actions of C have been listed in section V; their interpretations are found in Table 8. The compile-time state is minimal since it only needs to handle control flow. Its components are: a partially-generated program $\pi = [0: st_0, \ldots, k: st_k]$, a stack to hold parameters of target instructions, and a stack of origins of streams (the top is the origin of the current stream). Catenation of item a to the top of stack s is written as a•s.

Table 8. Interpretation of Compiling Algebra C

Domains:

Store = program x p* x l* (notation of Table 4)

Answer = program

Elementary Actions:

Notation: π = program, i = instruction

l = label

upd(π, l, i) = π, with instruction i added to the end of its l'th stream

θaddcode(i) c (π, p*, l . l*)
= c (upd(π, l, i), p*, l • l*)

θaddcode$_n$(i) c (π, p_n • ... • p_n • p*, p*, l • l*)
= c (upd(π, l, i($p_1, \ldots, p_n$)), p*, l • l*)

θpush(p) c (π, p*, l*) = c (π, p • p*, l*)

θneworig c (π, p*, l*) = c (π, p*, (k+1) • l*)
where $\pi = [0: str_0 \ldots k: str_k]$

θgoback c (π, p*, k • l • l*) = c (upd(π, k, goto(l + i)), p*, k • l • l*)
where i = length of the l'th stream of π

θoldorig c (π, p*, l • l*) = c (π, l • p*, l*)

Final Continuation: c_∞ (π, p*, l*) = π

Initial Store: st_0 = (emptyprogram, emptystack, 0)

VII CONCLUSIONS AND FUTURE WORK

Practical Remarks

Summarized in operational terms, our method is used as follows:

1. A semantic definition $\mathcal{S}$ is written mapping parse trees into IS (e.g. Table 2).

2. This is transformed into a "compiling semantics" $\mathcal{S}'$ associating with each parse tree a sequence of compile-time actions (e.g. Table 5). This is done by applying the "cg" function of Table 7 to the right sides of $\mathcal{S}$ equations.

3. A source program π can be compiled by using $\mathcal{S}'$ to see which compile-time actions are appropriate for π, and performing them (e.g. Table 6).

4. Meanings of the elementary actions of $\mathcal{S}$ can be specified by an interpretation I such as that of Table 3. The target program compiled from π will contain these elementary actions as instructions, with labels and atoms as parameters.

5. Consequently target programs can be made executable on a computer by writing routines (e.g. macros) for the elementary actions which are consistent with the interpretation I. This seems to be straightforward in practice.

Step 3 above has been described as though one began with the entire parse tree, built all of $\mathcal{S}'\pi$, and then evaluated $\mathcal{S}'\pi$ via Icom. Clearly $\mathcal{S}'\pi$ can be evaluated while it is being formed since it is constructed in left-to-right order. Further, it often happens that in $\mathcal{S}$ the subexpression references in semantic equations occur in the same order as in the productions. In this case $\mathcal{S}'$ is essentially a "simple syntax-directed translation" [AhU73], and compilation can be performed during top-down parsing, e.g. as is typically done in handwritten recursive descent compilers. The implications for automatic production of efficient one-pass compilers should be obvious

The target programs produces are minimal, containing only the elementary actions given in the semantics and a few gotos. Consequently it should be possible to produce reasonably efficient machine code, either by macro expansion as in step 5 or by more sophisticated special-purpose methods (e.g. peephole optimization, or writing a code generator which maintains a compile-time description of the run-time store). Another promising possibility is automatically to produce the code generator by applying similar algebraic methods to the terms used to describe stores and the elementary actions on them (e.g. an automatic analysis of Table 3 or Table 8).

The approach is clearly well suited to "bootstrapping" since the compiling algebra C is a subalgebra of IS. The second author has constructed a semantics which maps a set of semantic equations into the corresponding compiling semantics

(by applying Table 7), and has implemented its elementary actions as well as those of Icom using the PROLOG language. The result can be used to translate a semantics into a compiler in flowchart form with instructions "addcode" etc. This compiler can then be used to translate source programs into flowchart form in a quite efficient way.

The whole system could thus be "ported" to a new machine simply by writing routines for the elementary compiling and compiler generation actions.

Directions for Future Work

Much work remains to be done to develop a realistic compiler generation system using this approach. Following are some open problems.

1. To use the method all run time data structures (e.g. stacks, displays, etc.) must be specified in the interpretation. Methods to transform more abstract definitions into imperative semantics need to be investigated. A promising direction is to extend IS to include, for example, the binding semantic algebra BBS of [Mos81]. It appears to be possible to synthesize necessary stack components of the store (e.g. temporary results and return addresses) directly from $\mathfrak{S}$ for at least a reasonably large subalgebra of BBS.

2. A general method to isolate the evaluation of static information (e.g. symbol tables) and move it into the compiler needs to be developed. This has mostly been done by ad hoc methods until now, although [Gan80] is an exception. It appears likely that flow analysis methods (e.g. [MuJ81]) can be applied to this problem.

3. A general method for transforming definitions of elementary actions into code generation modules should be developed.

Finally, it should be mentioned that the method cannot handle all programming language constructs. Examples include self-modification, concurrency and parallelism.

Acknowledgements

The second author would like to thank IBM (Japan) and Kyoto University (particularly Professor Takasu) for making it possible to visit Japan to participate in two symposia and to meet many from the Japanese research community in computer science. This paper was written partly in response to the interest shown in the topic by Japanese researchers.

Discussions with Peter Mosses, Gordon Plotkin, Flemming Nielson, Brian Mayoh, Hanne Riis and others at the University of Aarhus have been very helpful during the development of these ideas.

REFERENCES

ADA80 Formal definition of the ADA programming language, preliminary version for public review, Honeywell Inc., Cii Honeywell Bull, INRIA France (1980).

ADJ77 Goguen, J.A., J.W. Thatcher, E.G. Wagner and J.B. Wright, Initial algebra semantics and continuous algebras, JACM 24, 68-95 (1977).

ADJ78 Goguen, J.A., J.W. Thatcher, E.G. Wagner, An initial algebra approach to the specification, correctness and implementation of abstract data types, in Current Trends in Programming Methodology vol IV (ed. R.T. Yeh) Prentice-Hall, Englewood Cliffs, N.J. (1978).

ADJ79 Thatcher, J.W., E.G. Wagner, J.B. Wright, More on advice on structuring compilers and proving them correct, Proc. 6th ICALP, Springer LNCS 74, Berlin (1979).

AhU73 Aho, A.V., J.D. Ullman, The Theory of Parsing, Translation and Compiling vol. II, Prentice-Hall, Englewood Cliffs, N.J. (1973).

Bjø77 Bjørner, D. Formal development of interpreters and compilers, Danmarks Tekniske Højskole ID673 (1977).

BuL69 Burstall, R.M., P.J. Landin, Programs and their proofs: an algebraic approach, Machine Intelligence 4 (1969).

Cat79 Cattel, R.G.G., J.M. Newcomer, B.W. Leverett, Code generation in a machine-independent compiler, SIGPLAN Notices, vol. 14, no. 8 (1979).

Chr81 Christiansen, H. A New Approach to Compiler Generation, Master's thesis in computer science, Aarhus University (1981).

Don79 Donegan, M.K., R.E. Noonan, S. Feycock, A code generator language, SIGPLAN Notices, vol. 14, no. 8 (1979).

EKM79 Eriksen, S.H., B.B. Jensen, B.B. Kristensen, O.L. Madsen, The BOBS-system, Aarhus University, DAIMI report PB-71 (1979).

Ers78 Ershov, A.P. On the essence of computation, in Formal Description of Programming Language Concepts, ed. E.J. Neuholdt, North-Holland (1978).

Gan77 Ganzinger, H., K. Ripken, R. Wilhelm, Automatic generation of optimizing multipass compilers, in Proc. IFIP Congress 77, Toronto, ed. B. Gillchrist, pp. 535-540, New York: North-Holland (1977).

Gan80 Ganzinger, H. Transforming denotational semantics into practical attribute grammars, in [Jon80], pp. 1-69.

Gau80 Gaudel, M.C. Specification of compilers as abstract data type representations, in [Jon80], pp. 140-164.

Gog78 Goguen, J.A. Order sorted algebras: exceptions and error sorts, coercions and overloaded operators, Semantics and Theory of Comp. Rpt. 14, UCLA (1978).

Gor79 Gordon, M.J.C. The Denotational Description of Programming Languages, An Introduction. Springer Verlag, Berlin (1979).

Joh74 Johnson, S.C. YACC - yet another compiler compiler, CSTR32, Bell Laboratories, Murray Hill, N.J.

Jon80 Jones, N.D. (editor) Semantics-Directed Compiler Generation. Lecture Notes in Computer Science 94, Springer-Verlag, Berlin (1980).

JoS80 Jones, N.D., D.A. Schmidt, Compiler generation from denotational semantics. Found in [Jon80], pp. 70-93.

Knu68 Knuth, D.E. Semantics of context-free languages, Math. Syst. Theory 2, no. 2, pp. 127-145 (1968).

Lor75 Lorho, B. Semantic attributes processing in the system DELTA, in Methods of Algorithmic Language Implementation (C.H.A. Koster ed.), pp. 21-40, Springer LNCS 47, Berlin (1977).

Luc69 Lucas, P., K. Walk, On the formal description of PL/I, Annual Review in Automatic Programming, vol. 6, pt. 3, Pergamon Press (1969).

McK70 McKeeman, W.M., J.J. Horning, D.B. Wortman, A Compiler Generator, Prentice-Hall, Englewood Cliffs, N.J. (1970).

McP67 McCarthy, J., J. Painter, Correctness of a compiler for arithmetic expressions, Proc. Symp. Appl. Math. 19, pp. 33-41 (1967).

MiS76 Milne, R.W., C. Strachey, A Theory of Programming Language Semantics, Chapman and Hall (UK), John Wiley (USA) (1976).

Mor73 Morris, F.L. Advice on structuring compilers and proving them correct, Proc. ACM Symp. Prin. Prog. Langs., Boston, pp. 144-152 (1973).

Mos74 Mosses, P. The Mathematical semantics of Algol 60. PRG-12, Oxford University Programming Research Group (1974).

Mos78 Mosses, P. Modular denotational semantics, draft paper, Computer Science Department, Aarhus University (1978).

Mos79 Mosses, P. SIS - Semantics Implementation System, Reference Manual and Users Guide. Aarhus University, DAIMI MD-30 (1979).

Mos80 Mosses, P. A constructive approach to compiler correctness. Found in: [Jon80], pp. 189-210.

Mos81 Mosses, P. A semantic algebra for binding constructs. In Formalization of Programming Concepts, LNCS 107, Springer-Verlag, pp. 408-418 (1981).

MuJ81 Muchnick, S., N.D. Jones, Program Flow Analysis, Prentice-Hall, Englewood Cliffs, N.J. (1981).

Oll75 Ollongren, A. A Definition of Programming Languages by Interpreting Automata, Academic Press (1975).

Plo81 Plotkin, G. A Structural Approach to Operational Semantics, Lecture notes, Computer Science Department, Aarhus University (1981).

Ras80 Raskovsky, M., P. Collier, From standard to implementation denotational semantics. Found in [Jon80], pp. 94-139.

RiM81 Riis, H., M. Madsen, The NEATS System, Computer Science Department, Aarhus University (1981).

Räi80 Räihä, K.-J. Experiences with the compiler writing system HLP, in [Jon80], pp. 350-362.

San75 Sandewall, E. Ideas about management of LISP data bases, 4th IJCAI, Tbilisi, vol. 2, pp. 585-592 (1975).

Sco76 Scott, D. Data types as lattices, SIAM Journal on Computing, vol. 5, no. 3, pp. 522-587 (1976).

ScS71 Scott, D., C. Strachey, Towards a mathematical semantics for computer languages, in Proc. Symp. "Computer and Automata", MRI Symposia, vol. XXI, Polytechnic Inst. of Brooklyn, N.Y. (1971).

Sto77 Stoy, J.E. Denotational semantics: The Scott-Strachey approach to programming language theory. MIT Press (1977).

Ten76 Tennent, R.D. The denotational semantics of programming languages, CACM vol. 19, no. 8 (1976).

Tur80 Turchin, V.F. Semantic definitions in REFAL and the automatic production of compilers, in [Jon80], pp. 441-474.

Wan80a Wand, M. Deriving target code as a representation of continuation semantics. Indiana University, Comp. Sc. Dept., Technical Report no. 94 (1980).

Wan80b Wand, M. Different advice on structuring compilers and proving them correct. Indiana University, Comp. Sc. Dept., Technical Report no. 95 (1980).